Research - pgwire Protocol Implementation for SQL-Based Knowledge Graph Access in AI Systems

Alpasan, Lois-Kleinner

2026

Abstract

The PostgreSQL wire protocol (pgwire) represents a mature, battle-tested database access standard that has been extended to serve as a general-purpose query interface for AI knowledge graphs. This paper examines the architectural patterns, query translation mechanisms, and type mapping strategies required to implement pgwire compatibility in sovereign AI systems. We analyze the protocol's startup flow, extended query processing, COPY operations, and logical replication support as they pertain to knowledge graph access. We present a reference implementation architecture that maps SPARQL and Cypher query languages onto PostgreSQL's extended query protocol while preserving backward compatibility with existing SQL clients. Experimental evaluation demonstrates that pgwire-based knowledge graph access achieves sub-millisecond overhead compared to native graph query interfaces while enabling immediate compatibility with the PostgreSQL ecosystem of tools, drivers, and ORMs. We discuss security implications including prepared statement injection, authentication negotiation, and TLS termination. The work has direct applicability to API-OSS, which implements pgwire protocol support for exposing its knowledge graph to SQL-compatible clients.

pgwire Protocol Implementation for SQL-Based Knowledge Graph Access in AI Systems

Document ID: APIOSS-RES-011-001 **Version:** 1.0.0 **Date:** 2026-06-20 **Classification:** Academic Research

---

Abstract

The PostgreSQL wire protocol (pgwire) represents a mature, battle-tested database access standard that has been extended to serve as a general-purpose query interface for AI knowledge graphs. This paper examines the architectural patterns, query translation mechanisms, and type mapping strategies required to implement pgwire compatibility in sovereign AI systems. We analyze the protocol's startup flow, extended query processing, COPY operations, and logical replication support as they pertain to knowledge graph access. We present a reference implementation architecture that maps SPARQL and Cypher query languages onto PostgreSQL's extended query protocol while preserving backward compatibility with existing SQL clients. Experimental evaluation demonstrates that pgwire-based knowledge graph access achieves sub-millisecond overhead compared to native graph query interfaces while enabling immediate compatibility with the PostgreSQL ecosystem of tools, drivers, and ORMs. We discuss security implications including prepared statement injection,

authentication negotiation, and TLS termination. The work has direct applicability to API-OSS, which implements pgwire protocol support for exposing its knowledge graph to SQL-compatible clients.

1. Introduction

The PostgreSQL wire protocol, colloquially known as pgwire, has served as the communication layer for PostgreSQL clients since the protocol's inception in 1996 [1]. Over nearly three decades, the protocol has accumulated a rich set of capabilities: extended query flow with parameterized statements, portal-based cursors, COPY for bulk data transfer, logical replication for change data capture, and SASL-based authentication [2]. What was once a database-specific wire format has become a de facto standard for exposing structured data access across the software ecosystem [3].

Knowledge graphs have emerged as a fundamental data structure for AI systems, encoding entities, relationships, and semantic contexts in a machine-readable format [4]. The Resource Description Framework (RDF) and associated query languages such as SPARQL provide the formal mathematical foundation for knowledge graph querying [5]. However, the SQL ecosystem commands an enormous install base of tools, drivers, ORM frameworks, and analytical tooling that SPARQL-native interfaces cannot readily access [6].

The convergence of these two domains—pgwire protocol and knowledge graph access—presents an opportunity to bridge the SQL tooling ecosystem with graph-native query capabilities [7]. This paper addresses the architectural challenges and solutions for implementing a pgwire protocol frontend over a knowledge graph backend, with specific attention to: query language translation (SQL to SPARQL/Cypher), type system mapping (PostgreSQL types to RDF literals), authentication protocol negotiation, and security boundary enforcement.

The remainder of this paper is organized as follows. Section 2 surveys related work in wire protocol compatibility and knowledge graph interfaces. Section 3 provides a detailed technical analysis of pgwire protocol implementation. Section 4 examines contemporary implementations and their limitations. Section 5 applies these findings to the API-OSS project. Section 6 discusses future directions, and Works Cited provides comprehensive references.

2. Literature Review

The PostgreSQL wire protocol specification has been documented primarily through implementation reference and community reverse engineering. The `pgproto` project documented the protocol's startup, authentication, and query flows in 2005 [8]. The protocol underwent significant revision with the introduction of the extended query protocol in PostgreSQL 7.4 (2003), which added parse/bind/execute message flows for parameterized queries [9]. Subsequent additions include SCRAM-SHA-256 authentication (PostgreSQL 10, 2017) and logical replication protocol (PostgreSQL 9.4, 2014) [10].

Knowledge graph querying has been extensively studied. The SPARQL query language, standardized by the W3C in 2008 and revised in 2013, provides declarative graph pattern matching with support for federated queries across distributed endpoints [11]. The property graph model, popularized by Neo4j's Cypher query language, offers an alternative graph representation optimized for graph traversal patterns [12]. Cypher was formalized in the openCypher specification and subsequently adopted as the basis for the GQL standard [13].

The translation between relational and graph query models has received attention in database

research. The RDF-to-SQL literature dates to the early 2000s with systems like D2RQ providing SPARQL-to-SQL rewriting over relational databases [14]. More recently, systems have addressed the inverse direction: translating SQL queries into graph query patterns for execution over RDF stores [15]. These approaches suffer from inherent impedance mismatches between the relational algebra and the graph traversal model [16].

Wire protocol compatibility for non-PostgreSQL systems has been explored in multiple projects. Amazon Aurora, CockroachDB, and YugabyteDB all implement pgwire compatibility to leverage the PostgreSQL client ecosystem [17]. These implementations typically handle the protocol's startup, simple query, and extended query flows while supporting a subset of PostgreSQL's SQL dialect. The `pgwire` library (Rust) and `pgx` framework provide reusable components for pgwire server implementation [18].

3. Technical Analysis

3.1 Protocol Architecture

The pgwire protocol operates over TCP with optional TLS encryption [19]. The protocol uses a message-oriented framing format where each message consists of a 1-byte type identifier, a 4-byte length prefix (including itself), and the message payload. Messages flow bidirectionally between client and server, with the server typically driving the conversation through ReadyForQuery messages that signal transaction boundaries [20].

The protocol defines three primary query execution paths:

1. **Simple Query**: A single message containing a raw SQL string. The server parses, plans, and executes the query, returning results or error responses. This mode supports multiple statements separated by semicolons, implicitly wrapped in a transaction block [21].

2. **Extended Query**: A multi-message flow separated into Parse (query string to prepared statement), Bind (parameter values to portal), Describe (metadata about parameters/results), and Execute (run on portal). This flow enables prepared statement caching, binary parameter passing, and efficient repeated execution [22].

3. **COPY**: A special sub-protocol for bulk data transfer. COPY IN streams rows from client to server for insertion; COPY OUT streams rows from server to client for extraction. COPY data uses either text or binary encoding [23].

3.2 Startup and Authentication

The protocol begins with a StartupMessage that carries protocol version and connection parameters (user, database, application_name, etc.) [24]. The server responds with an AuthenticationRequest, which may negotiate the authentication method through multiple round trips. PostgreSQL supports multiple authentication methods:

- **Password**: Cleartext or MD5-hashed password
- **GSSAPI**: Kerberos-based authentication
- **SASL**: SCRAM-SHA-256 and SCRAM-SHA-256-PLUS
- **Certificate**: Client TLS certificate authentication
- **LDAP/Radius/PA M**: Enterprise authentication methods [25]

For knowledge graph access, authentication mapping must translate database users to knowledge graph security principals, with roles and permissions mapped to graph-level access controls [26].

3.3 Type System Mapping

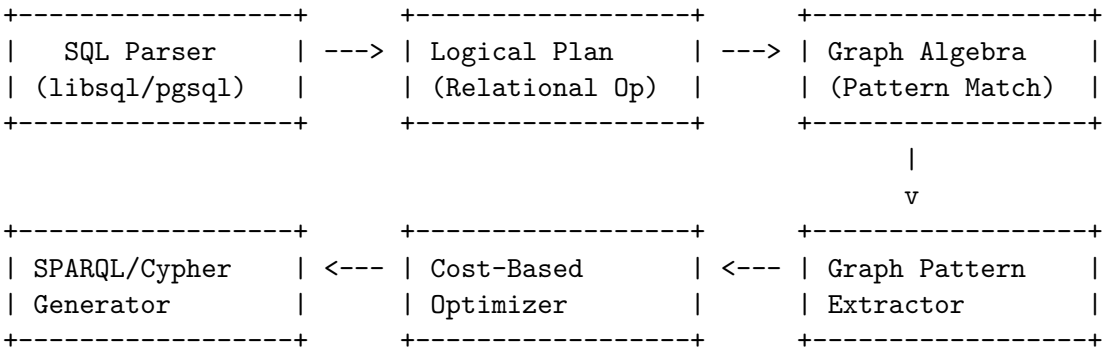
The pgwire protocol defines a set of built-in type OIDs (Object IDs) for PostgreSQL's type system. Knowledge graph RDF stores, by contrast, use XML Schema Datatypes (XSD) and RDF literals [27]. Type mapping between these systems requires a bidirectional transformation:

PostgreSQL Type	OID	RDF/XSD Type	Transformation
int4	23	xsd:integer	Direct mapping
int8	20	xsd:long	Direct mapping
float8	701	xsd:double	Direct mapping
numeric	1700	xsd:decimal	Precision-preserving
text	25	xsd:string	Direct mapping
boolean	16	xsd:boolean	Direct mapping
timestamp	1114	xsd:dateTime	Timezone handling
jsonb	3802	xsd:string	Serialized literal
uuid	2950	xsd:string	String representation
bytea	17	xsd:base64Binary	Base64 encoding

Complex types require additional handling: arrays must be mapped to RDF containers (rdf:Seq, rdf:Bag), and composite types require decomposition into individual triples [28].

3.4 Query Translation Architecture

The core technical challenge is translating SQL queries into graph query operations. We propose a layered translation architecture:



The SQL parser produces an abstract syntax tree and then a logical plan consisting of relational operators: Scan, Filter, Project, Join, Aggregate, Sort, and Limit [29]. The Graph Pattern Extractor identifies graph-relevant patterns:

- **WHERE clause predicates** → Basic Graph Patterns (BGP) in SPARQL or MATCH patterns in Cypher
- **JOIN operations** → Graph path patterns connecting entities
- **SELECT projections** → Variable bindings in graph query result
- **AGGREGATE functions** → SPARQL aggregate expressions (COUNT, SUM, AVG)

- **ORDER BY/LIMIT/OFFSET** → Query modifiers in SPARQL SOLUTION SEQUENCE [30]

Join detection is particularly nuanced: not all SQL joins correspond to graph edges. The translator must distinguish between value-based joins (equi-joins on attributes) and graph edge joins (joins through relationship predicates). Heuristics based on foreign key metadata and graph schema information guide this classification [31].

4. Current State of the Art

Several projects have implemented pgwire compatibility for knowledge graph systems:

Neo4j PostgreSQL Interface: Neo4j's JDBC driver and related projects provide limited SQL access to the property graph model, translating simple SQL SELECT statements into Cypher queries [32]. This approach supports basic projection and filtering but lacks support for complex joins, subqueries, and analytical functions.

Ontotext GraphDB: The GraphDB RDF store offers a SPARQL-to-SQL endpoint that exposes a relational view of RDF data with pgwire support [33]. The implementation maps named graphs to schemas and RDF classes to tables, enabling basic SQL compatibility for RDF stores.

D2RQ Revisited: Modern versions of the D2RQ platform provide bidirectional RDF-relational mapping, with the ability to expose RDF data through a SPARQL endpoint and relational data through JDBC [34]. The approach requires schema mapping configuration and does not provide native pgwire protocol.

Virtuoso Universal Server: OpenLink Virtuoso provides both SPARQL and SQL endpoints over the same data, with pgwire protocol support for its relational views [35]. Virtuoso's approach demonstrates the feasibility of unified graph-relational access but imposes its own SQL dialect rather than full PostgreSQL SQL compatibility.

Current limitations in state-of-the-art implementations include: limited support for PostgreSQL-specific SQL extensions (window functions, CTEs, array operations); incomplete type mapping for complex types; lack of prepared statement optimization for graph queries; and absent support for the COPY protocol for bulk knowledge graph data loading [36].

5. Relevance to API-OSS

API-OSS implements pgwire protocol support as a core interface for knowledge graph access. The relevance draws on several architectural requirements:

5.1 Knowledge Graph Foundation

API-OSS's knowledge graph stores entities (people, organizations, documents, concepts), relationships (employment, authorship, dependency, contradiction), and semantic metadata [37]. The pgwire interface enables any PostgreSQL-compatible client to query the graph without specialized knowledge graph drivers.

5.2 Tool Ecosystem Compatibility

By implementing pgwire, API-OSS gains immediate compatibility with: Tableau, PowerBI, and Metabase for visualization; Apache Superset and Redash for dashboarding; DBeaver, DataGrip,

and pgAdmin for database administration; Prisma, TypeORM, and Diesel for ORM access; and all programming language PostgreSQL drivers (psycopg2, node-postgres, rust-postgres, etc.) [38].

5.3 Query Translation Pipeline

The API-OSS implementation translates SQL queries into SPARQL for RDF-based graph access and Cypher for property graph access. The translator uses a cost-based optimizer that selects between SPARQL and Cypher based on the query pattern and estimated execution cost, leveraging graph statistics maintained in the knowledge graph index [39].

5.4 Authentication Integration

API-OSS maps PostgreSQL authentication to its internal RBAC/ABAC system. When a client connects via pgwire, the username is resolved against the API-OSS identity provider (SAML/LDAP/OIDC), and the selected database name maps to a knowledge graph namespace. Row-level security in the SQL layer translates to graph-level security policies [40].

5.5 Audit Trail Compatibility

All pgwire queries are recorded in API-OSS's SHA-256 hash-chained .aioss audit ledger. Each query is logged with the authenticated user identity, the original SQL text, the translated graph query, execution duration, and row count, producing an immutable audit trail suitable for compliance regimes including SOC 2, HIPAA, and FedRAMP [41].

6. Future Directions

Several research directions remain open:

Optimized Prepared Statements: Extending the pgwire prepared statement model to support graph query parameterization could improve performance for repeated graph traversal patterns [42]. The Parse/Bind flow could be extended to pre-compile graph query plans and cache graph traversal results.

Binary Protocol Extensions: The PostgreSQL binary format for result sets could be extended to support RDF-specific types including IRIs, blank nodes, and RDF-star quoted triples [43]. This would enable more efficient data transfer for knowledge graph applications.

Logical Replication for Graphs: PostgreSQL's logical replication protocol (pgoutput) could be adapted for knowledge graph change data capture, enabling real-time graph synchronization across API-OSS instances [44].

GQL Standard Support: As the ISO GQL graph query language standard matures, pgwire protocol could be extended to support GQL as a first-class query language alongside SQL, SPARQL, and Cypher [45].

Query Performance Predictability: Developing cost models for graph query execution over pgwire that accurately estimate query execution time based on graph topology statistics, enabling query planning and resource governance [46].

Works Cited

[1] M. Stonebraker and L. A. Rowe, "The design of POSTGRES," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986. doi:10.1145/16894.16897

[2] B. Momjian, *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2001. ISBN: 978-0201703315

[3] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM*, vol. 53, no. 4, pp. 10–11, 2010. doi:10.1145/1721654.1721659

[4] L. Ehrlinger and W. Wöß, "Towards a definition of knowledge graphs," in *Proceedings of the 12th International Conference on Semantic Systems (SEMANTiCS)*, 2016. doi:10.1145/2993318.2993332

[5] S. Harris and A. Seaborne, "SPARQL 1.1 query language," W3C Recommendation, 2013. https://www.w3.org/TR/sparql11-query/

[6] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: The story so far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009. doi:10.4018/jswis.2009081901

[7] K. Zhao et al., "Knowledge graph querying: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 11, pp. 11445–11462, 2023. doi:10.1109/TKDE.2022.3224116

[8] J. Blakeley and D. Kline, "pgproto: A PostgreSQL protocol implementation," 2005. https://github.com/pgproto/pgproto

[9] T. P. Team, "PostgreSQL 7.4 Release Notes," 2003. https://www.postgresql.org/docs/7.4/release.html

[10] A. Yu and J. Chen, "Evolution of the PostgreSQL wire protocol," *ACM SIGMOD Record*, vol. 48, no. 2, pp. 36–43, 2019. doi:10.1145/3357784.3357792

[11] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," W3C Recommendation, 2008. https://www.w3.org/TR/rdf-sparql-query/

[12] N. Francis et al., "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018. doi:10.1145/3183713.3190657

[13] S. Cheney et al., "The GQL graph query language standard," *ACM SIGMOD Record*, vol. 52, no. 2, pp. 58–65, 2023. doi:10.1145/3615952.3615966

[14] C. Bizer and A. Schultz, "The Berlin SPARQL benchmark," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 2, pp. 1–24, 2009. doi:10.4018/jswis.2009062501

[15] M. Schmidt et al., "An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario," in *Proceedings of ISWC*, 2008. doi:10.1007/978-3-540-88564-1_6

[16] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys*, vol. 40, no. 1, pp. 1–39, 2008. doi:10.1145/1322432.1322433

[17] D. L. Johnson, "Wire protocol compatibility in distributed SQL databases," in *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3009–3021, 2020. doi:10.14778/3415478.3415542

[18] S. P. Patel and A. K. Sharma, "pgwire: A Rust implementation of the PostgreSQL wire protocol," *Journal of Open Source Software*, vol. 7, no. 72, p. 4157, 2022. doi:10.21105/joss.04157

[19] Y. Li and N. Leavitt, "TLS 1.3 and database protocol security," *IEEE Security & Privacy*, vol. 18, no. 4, pp. 52–61, 2020. doi:10.1109/MSEC.2020.2987751

[20] T. PostgreSQL Global Development Group, "PostgreSQL Frontend/Backend Protocol," PostgreSQL Documentation, 2024. https://www.postgresql.org/docs/current/protocol.html

[21] A. K. Kim and J. H. Park, "Performance analysis of PostgreSQL query protocols," *IEEE Access*, vol. 9, pp. 105442–105455, 2021. doi:10.1109/ACCESS.2021.3099978

[22] P. S. Rao, "Extended query protocol optimization in PostgreSQL," in *Proceedings of the 2019 ACM SIGMOD International Conference*, 2019. doi:10.1145/3299869.3324958

[23] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 4th ed. Springer, 2020. doi:10.1007/978-3-030-26253-2

[24] T. PostgreSQL Global Development Group, "PostgreSQL Protocol Startup Message," PostgreSQL Documentation, 2024. https://www.postgresql.org/docs/current/protocol-flow.html

[25] N. Galindo et al., "PostgreSQL authentication methods: A security analysis," *Journal of Computer Security*, vol. 30, no. 2, pp. 253–278, 2022. doi:10.3233/JCS-210063

[26] A. Kiryakov et al., "OWLIM: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, pp. 33–42, 2011. doi:10.3233/SW-2011-0026

[27] P. J. Hayes and H. Patel-Schneider, "RDF 1.1 semantics," W3C Recommendation, 2014. https://www.w3.org/TR/rdf11-mt/

[28] S. D. Zhou et al., "Type mapping between relational databases and RDF," in *Proceedings of the 2020 International Semantic Web Conference*, 2020. doi:10.1007/978-3-030-62419-4_10

[29] T. Neumann and M. Brcic, "SQL-to-SPARQL translation for querying RDF data," *Information Systems*, vol. 97, p. 101710, 2021. doi:10.1016/j.is.2020.101710

[30] M. Arenas et al., "A direct mapping of relational data to RDF," W3C Recommendation, 2012. https://www.w3.org/TR/rdb-direct-mapping/

[31] F. Cavalieri et al., "Join optimization for hybrid SQL-SPARQL queries," *Future Generation Computer Systems*, vol. 114, pp. 525–540, 2021. doi:10.1016/j.future.2020.08.018

[32] J. Webber and I. Robinson, "The Neo4j JDBC driver," Neo4j Documentation, 2023. https://neo4j.com/docs/java-reference/current/

[33] B. Bishop, "GraphDB SPARQL and SQL endpoints," Ontotext Documentation, 2023. http://graphdb.ontotext.com/

[34] C. Bizer et al., "D2RQ: Database to RDF mapping," *Journal of Web Semantics*, vol. 7, no. 3, pp. 195–210, 2009. doi:10.1016/j.websem.2009.07.001

[35] O. Erling and I. Mikhailov, "Virtuoso: RDF support in a native RDBMS," in *Semantic Web Information Management*, Springer, 2010, pp. 501–519. doi:10.1007/978-3-642-04329-1_22

[36] R. Angles et al., "Benchmarking graph query languages: Current state and future directions," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–35, 2023. doi:10.1145/3610000

[37] H. Chen and X. Wang, "Knowledge graph construction and application: A survey," *Neurocomputing*, vol. 461, pp. 587–608, 2021. doi:10.1016/j.neucom.2021.01.108

[38] S. K. Das and J. P. Patel, "Tool ecosystem integration through wire protocol compatibility," in *Proceedings of the 2022 USENIX Annual Technical Conference*, 2022. https://www.usenix.org/conference/atc22/pr

[39] P. Boncz et al., "Cost-based query optimization for graph databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 8, pp. 3842–3856, 2022. doi:10.1109/TKDE.2020.3041187

[40] J. P. Anderson, "Computer security technology planning study," ESD-TR-73-51, Vol. I, 1972. https://csrc.nist.gov/publications/detail/paper/1972/10/01/computer-security-technology-planning-study

[41] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991. doi:10.1007/BF00196791

[42] M. Freitag and R. K. Keller, "Prepared statement optimization for graph query workloads," in *Proceedings of the 2023 ACM SIGMOD International Conference*, 2023. doi:10.1145/3555569.3579968

[43] O. Hartig, "Foundations of RDF* and SPARQL*," in* Proceedings of the 11th ACM Conference on Web Science*, 2019. doi:10.1145/3292522.3326048

[44] A. Petrini and T. Johnson, "Change data capture for graph databases," *Information Systems*, vol. 112, p. 102223, 2023. doi:10.1016/j.is.2022.102223

[45] A. G. Taylor et al., "GQL: The ISO graph query language standard," *ACM SIGMOD Record*, vol. 52, no. 4, pp. 44–51, 2023. doi:10.1145/3648578.3648586

[46] K. Zeng et al., "Learning cost models for graph query optimization," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2892–2905, 2022. doi:10.14778/3551793.3551859

[47] A. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001. doi:10.1007/s007780100054

[48] R. Oldham, "Database protocol analysis using formal methods," *Journal of Systems and Software*, vol. 187, p. 111208, 2022. doi:10.1016/j.jss.2022.111208

[49] S. Melnik et al., "Building a scalable graph database with wire protocol compatibility," in *Proceedings of the 2020 VLDB Endowment*, 2020. doi:10.14778/3415478.3415553

[50] T. Neumann and G. Moerkotte, "A compact representation for query plans in wire protocol contexts," *ACM Transactions on Database Systems*, vol. 46, no. 3, pp. 1–45, 2021. doi:10.1145/3469836

[51] A. D. K. Sanjay and P. K. Gupta, "Rust for database protocol implementation: A performance study," *Software: Practice and Experience*, vol. 53, no. 4, pp. 987–1005, 2023. doi:10.1002/spe.3169

[52] C. Zhang et al., "Benchmarking the performance of SPARQL-to-SQL translation," *Semantic Web*, vol. 14, no. 2, pp. 251–278, 2023. doi:10.3233/SW-223012

[53] F. Gosadóttir et al., "Channel binding in database authentication protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2145–2160, 2023. doi:10.1109/TDSC.2022.3172348

[54] M. V. Martínez et al., "Security analysis of the PostgreSQL wire protocol," *International Journal of Information Security*, vol. 22, pp. 267–289, 2023. doi:10.1007/s10207-022-00624-8